Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform

26.01.2014 (http://www.ethereum.org/ethereum.html)

In the last few months, there has been a great amount of interest into the area of using Bitcoin-like blockchains - the mechanism that allows for the entire world to agree on the state of a public ownership database - for more than just money. Commonly cited applications include using on-blockchain digital assets to represent custom currencies and financial instruments ("colored coins"), "smart property" devices such as cars which track a colored coin on a blockchain to determine their present legitimate owner, as well as more advanced applications such as decentralized exchange, financial derivatives, peer-to-peer gambling and on-blockchain identity and reputation systems. Perhaps the most ambitious of all cited applications is the concept of *autonomous agents* or *decentralized autonomous organizations* (DAOs) - autonomous entities that operate on the blockchain without any central control whatsoever, eschewing all dependence on legal contracts and organizational bylaws in favor of having resources and funds autonomously managed by a self-enforcing smart contract on a cryptographic blockchain.

However, most of these applications are difficult to implement today, simply because the scripting systems of Bitcoin, and even next-generation cryptocurrency protocols such as the Bitcoin-based colored coins protocol and so-called "metacoins", are far too limited to allow the kind of arbitrarily complex computation that DAOs require. What this project intends to do is take the innovations that such protocols bring, and generalize them - create a fully-fledged, Turing-complete (but heavily fee-regulated) cryptographic ledger that allows participants to encode arbitrarily complex contracts, autonomous agents and relationships that will be mediated entirely by the blockchain. Rather than being limited to a specific set of transaction types, users will be able to use Ethereum as a sort of "Lego of crypto-finance" - that is to say, one will be able to implement any feature that one desires simply by coding it in the protocol's internal scripting language. Custom currencies, financial derivatives, identity systems and decentralized organizations will all be easy to do, but more importantly, unlike previous systems, it will also be possible to construct transaction types that even the Ethereum developers did not imagine. Altogether, we believe that this design is a solid step toward the realization of "cryptocurrency 2.0"; we hope that Ethereum will be as significant an addition to the cryptocurrency ecosystem as the advent of Web 2.0 was to the static-content-only internet of 1999.

Table of Contents

Why A New Platform?

When one wants to create a new application, especially in an area as delicate as cryptography or cryptocurrency, the immediate, and correct, first instinct is to use existing protocols as much as possible. There is no need to create a new currency, or even a new protocol, when the problem can be solved entirely by using existing technologies. Indeed, the puzzle of attempting to solve the problems of smart property, smart contracts and decentralized autonomous corporations on top of Bitcoin is how our interest in next-generation cryptocurrency protocols originally started. Over the course of our research, however, it became evident that while the Bitcoin protocol is more than adequate for currency, basic multisignature escrow and certain simple versions of smart contracts, there are fundamental limitations that make it non-viable for anything beyond a certain very limited scope of features.

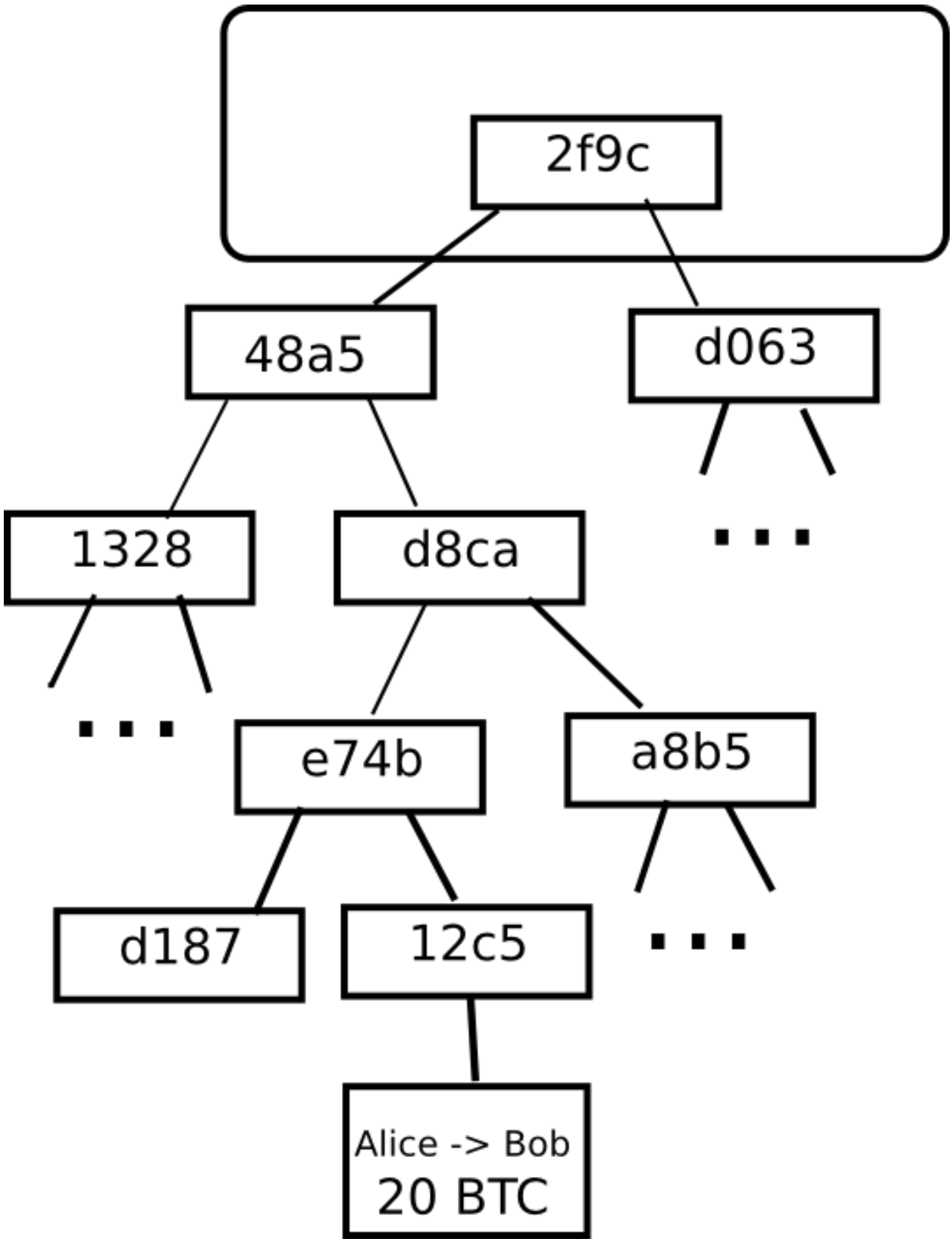
Colored Coins

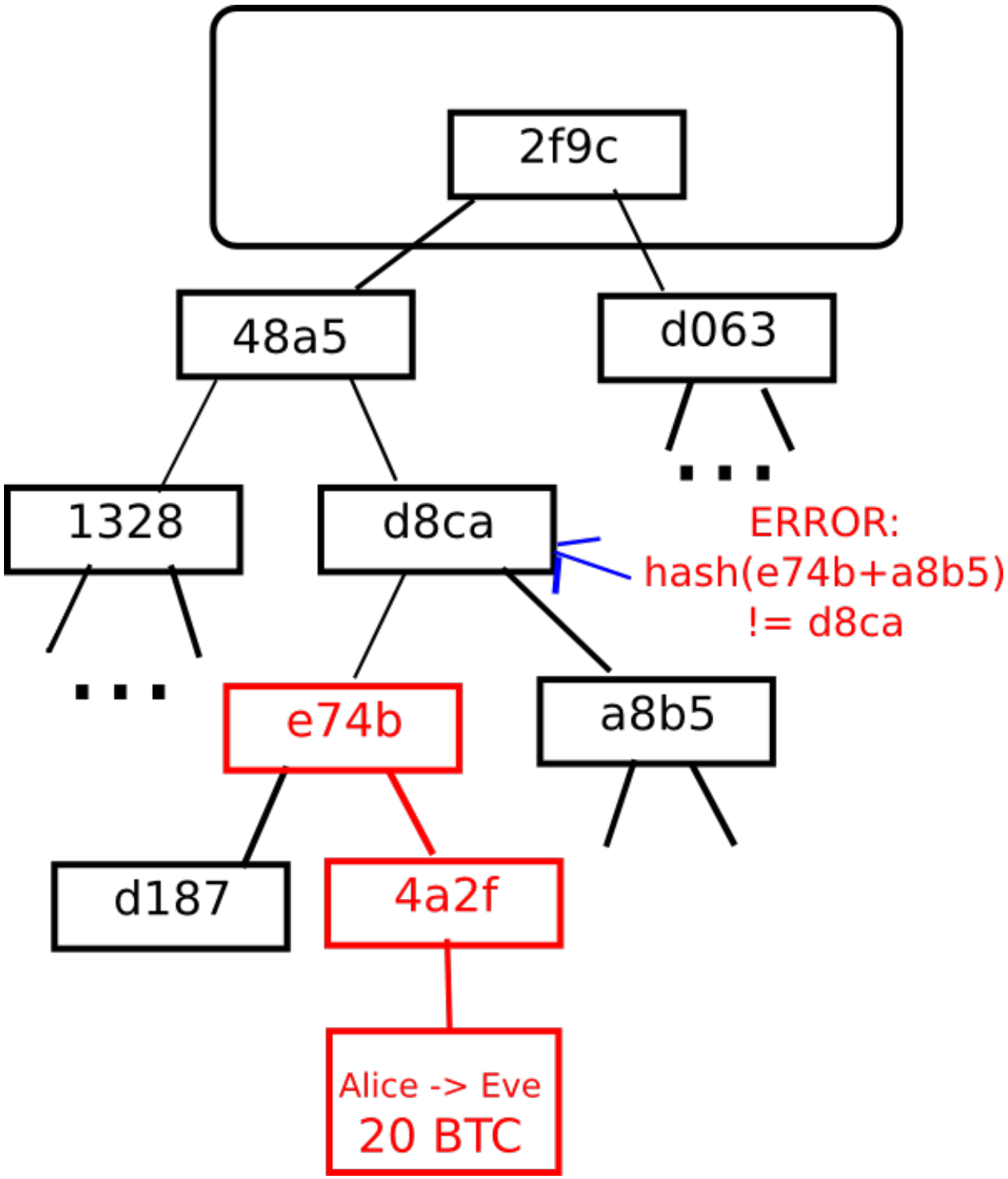
The first attempt to implement a system for managing smart property and custom currencies and assets on top of a blockchain was built as a sort of overlay protocol on top of Bitcoin, with many advocates making a comparison to the way that, in the internet protocol stack, HTTP serves as a layer on top of TCP. The colored coins protocol is roughly defined as follows:

1. A colored coin issuer determines that a given transaction output `H:i` (`H` being the transaction hash and `i` the output index) represents a certain asset, and publishes a "color definition" specifying this transaction output alongside what it represents (eg. 1 satoshi from `H:i` = 1 ounce of gold redeemable at Stephen's Gold Company)
2. Others "install" the color definition file in their colored coin clients.
3. When the color is first released, output `H:i` is the only transaction output to have that color.
4. If a transaction spends inputs with color X, then its outputs will also have color X. For example, if the owner of `H:i` immediately makes a transaction to split that output among five addresses, then those transaction outputs will all also have color X. If a transaction has inputs of different colors, then a "color transfer rule" or "color kernel" determines which colors which outputs are (eg. a very naive implementation may say that output 0 has the same color as input 0, output 1 the same color as input 1, etc).
5. When a colored coin client notices that it received a new transaction output, it uses a back-tracing algorithm based on the color kernel to determine the color of the output. Because the rule is deterministic, all clients will agree on what color (or colors) each output has.

However, the protocol has several fundamental flaws:

Simplified Payment Verification in Bitcoin


2f9c
48a5
d063
1328
d8ca
e74b
a8b5
d187
12c5
Alice -> Bob
20 BTC



Left: it suffices to present only a small number of nodes in a Merkle tree to give a proof of the validity of a branch.
Right: any attempt to change any part of the Merkle tree will eventually lead to an inconsistency somewhere up the chain.

1. **Difficulty of simplified payment verification** - Bitcoin's Merkle tree construction allows for a protocol known as "simplified payment verification", where a client that does not download the full blockchain can quickly determine the validity of a transaction output by asking other nodes to provide a cryptographic proof of the validity of a single branch of the tree. The client will still need to download the block headers to be secure, but the amount

of data bandwidth and verification time required drops by a factor of nearly a thousand. With colored coins, this is much harder. The reason is that one cannot determine the color of a transaction output simply by looking up the Merkle tree; rather, one needs to employ the backward scanning algorithm, fetching potentially thousands of transactions and requesting a Merkle tree validity proof of each one, before a client can be fully satisfied that a transaction has a certain color. After over a year of investigation, including help from ourselves, no solution has been found to this problem.

2. **Incompatibility with scripting** - as mentioned above, Bitcoin does have a moderately flexible scripting system, for example allowing users to sign transactions of the form "I release this transaction output to anyone willing to pay to me 1 BTC". Other examples include assurance contracts, efficient micropayments and on-blockchain auctions. However, this system is inherently not color-aware; that is to say, one cannot make a transaction of the form "I release this transaction output to anyone willing to pay me one gold coin defined by the genesis `H:i`", because the scripting language has no idea that a concept of "colors" even exists. One major consequence of this is that, while trust-free swapping of two different colored coins is possible, a full decentralized exchange is not since there is no way to place an enforceable order to buy or sell.
3. **Same limitations as Bitcoin** - ideally, on-blockchain protocols would be able to support advanced derivatives, bets and many forms of conditional transfers. Unfortunately, colored coins inherits the limitations of Bitcoin in terms of the impossibility of many such arrangements.

Metacoins

Another concept, once again in the spirit of sitting on top of Bitcoin much like HTTP over TCP, is that of "metacoins". The concept of a metacoin is simple: the metacoin protocol provides for a way of encoding metacoin transaction data into the outputs of a Bitcoin transaction, and a metacoin node works by processing all Bitcoin transactions and evaluating Bitcoin transactions that are valid metacoin transactions in order to determine the current account balances at any given time. For example, a simple metacoin protocol might require a transaction to have four outputs: `MARKER`, `FROM`, `TO` and `VALUE`. `MARKER` would be a specific marker address to identify a transaction as a metacoin transaction. `FROM` would be the address that coins are sent from. `TO` would be the address that coins are sent to, and `VALUE` would be an address encoding the amount sent. Because the Bitcoin protocol is not metacoin-aware, and thus will not reject invalid metacoin transactions, the metacoin protocol must treat all transactions with the first output going to `MARKER` as valid and react accordingly. For example, an implementation of the transaction processing part of the above described metacoin protocol might look like this:

```
if tx.output[0] != MARKER:
    break
else if balance[tx.output[1]] < decode_value(tx.output[3]):
    break
else if not tx.hasSignature(tx.output[1]):
```

```
        break
    else:
        balance[tx.output[1]] -= decode_value(tx.output[3]);
        balance[tx.output[2]] += decode_value(tx.output[3]);
```

The advantage of a metacoin protocol is that the protocol can allow for more advanced transaction types, including custom currencies, decentralized exchange, derivatives, etc, that are impossible to implement using the underlying Bitcoin protocol by itself. However, metacoins on top of Bitcoin have one major flaw: simplified payment verification, already difficult with colored coins, is outright impossible on a metacoin. The reason is that while one can use SPV to determine that there is a transaction sending 30 metacoins to address X, that by itself does not mean that address X has 30 metacoins. What if the sender of the transaction did not have 30 metacoins to start with and so the transaction is invalid? Ultimately, finding out any part of the current state requires scanning through all transactions since the metacoin's original launch to figure out which transactions are valid and which ones are not. This makes it impossible to have a truly secure client without downloading the entire, arguably prohibitively large, Bitcoin blockchain.

In both cases, the conclusion is as follows. The effort to build more advanced protocols on top of Bitcoin, like HTTP over TCP, is admirable, and is indeed the correct way to go in terms of implementing advanced decentralized applications. However, the attempt to build colored coins and metacoins on top of Bitcoin is more like building HTTP over SMTP. The intention of SMTP was to transfer email messages, not serve as a backbone for generic internet communications, and one would have had to implement many inefficient and architecturally ugly practices in order to make it effective. Similarly, while Bitcoin is a great protocol for making simple transactions and storing value, the evidence above shows that Bitcoin is absolutely not intended to function, and cannot function, as a base layer for financial peer-to-peer protocols in general.

Ethereum solves the scalability issues by being hosted on its own blockchain, and by storing a distinct "state tree" in each block along with a transaction list. Each "state tree" represents the current state of the entire system, including address balances and contract states. Ethereum contracts are allowed to store data in a persistent memory storage. This storage, combined with the Turing-complete scripting language, allows us to encode an entire currency inside of a single contract, alongside countless other types of cryptographic assets. Thus, the intention of Ethereum is not to replace the colored coins and metacoin protocols described above. Rather, Ethereum intends to serve as a superior foundational layer offering a uniquely powerful scripting system on top of which arbitrarily advanced contracts, currencies and other decentralized applications can be built. If existing colored coins and metacoin projects were to move onto Ethereum, they would gain the benefits of Ethereum's simplified payment verification, the option to be compatible with Ethereum's financial derivatives and decentralized exchange, and the ability to work together on a single network. With Ethereum, someone with an idea for a new contract or transaction type that might drastically improve the state of what can be done with cryptocurrency would not need

to start their own coin; they could simply implement their idea in Ethereum script code. In short, Ethereum is a foundation for innovation.

Philosophy

The design behind Ethereum is intended to follow the following principles:

1. **Simplicity** - the Ethereum protocol should be as simple as possible, even at the cost of some data storage or time inefficiency. An average programmer should ideally be able to follow and implement the entire specification, so as to fully realize the unprecedented democratizing potential that cryptocurrency brings and further the vision of Ethereum as a protocol that is open to all. Any optimization which adds complexity should not be included unless that optimization provides very substantial benefit.
2. **Universality** - a fundamental part of Ethereum's design philosophy is that Ethereum does not have "features". Instead, Ethereum provides an internal Turing-complete scripting language, which a programmer can use to construct any smart contract or transaction type that can be mathematically defined. Want to invent your own financial derivative? With Ethereum, you can. Want to make your own currency? Set it up as an Ethereum contract. Want to set up a full-scale Daemon or Skynet? You may need to have a few thousand interlocking contracts, and be sure to feed them generously, to do that, but nothing is stopping you with Ethereum at your fingertips.
3. **Modularity** - the parts of the Ethereum protocol should be designed to be as modular and separable as possible. Over the course of development, our goal is to create a program where if one was to make a small protocol modification in one place, the application stack would continue to function without any further modification. Innovations such as Dagger, Patricia trees and RLP should be implemented as separate libraries and made to be feature-complete even if Ethereum does not require certain features so as to make them usable in other protocols as well. Ethereum development should be maximally done so as to benefit the entire cryptocurrency ecosystem, not just itself.
4. **Agility** - details of the Ethereum protocol are not set in stone. Although we will be extremely judicious about making modifications to high-level constructs such as the C-like language and the address system, computational tests later on in the development process may lead us to discover that certain modifications to the algorithm or scripting language will substantially improve scalability or security. If any such opportunities are found, we will exploit them.
5. **Non-discrimination** - the protocol should not attempt to actively restrict or prevent specific categories of usage. All regulatory mechanisms in the protocol should be designed to directly regulate the harm and not attempt to oppose specific undesirable applications. A programmer can even run an infinite loop script on top of Ethereum for as long as they are willing to keep paying the per-computational-step transaction fee.

Basic Building Blocks

At its core, Ethereum starts off as a fairly regular memory-hard proof-of-work mined cryptocurrency without many extra complications. In fact, Ethereum is in some ways simpler than the Bitcoin-based cryptocurrencies that we use today. The concept of a transaction having multiple inputs and outputs, for example, is gone, replaced by a more intuitive balance-based model (to prevent transaction replay attacks, as part of each account balance we also store an incrementing nonce). Sequence numbers and lock times are also removed, and all transaction and block data is encoded in a single format. Instead of addresses being the RIPEMD160 hash of the SHA256 hash of the public key prefixed with `04`, addresses are simply the last 20 bytes of the SHA3 hash of the public key. Unlike other cryptocurrencies, which aim to offer a large number of "features", Ethereum intends to take features away, and instead provide its users with near-infinite power through an all-encompassing mechanism known as "contracts".

Modified GHOST Implementation

The "Greedy Heavist Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013. The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing stale blocks 70% of the time whereas B will have a risk of producing stale blocks 90% of the time. Thus, if the stale rate is high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

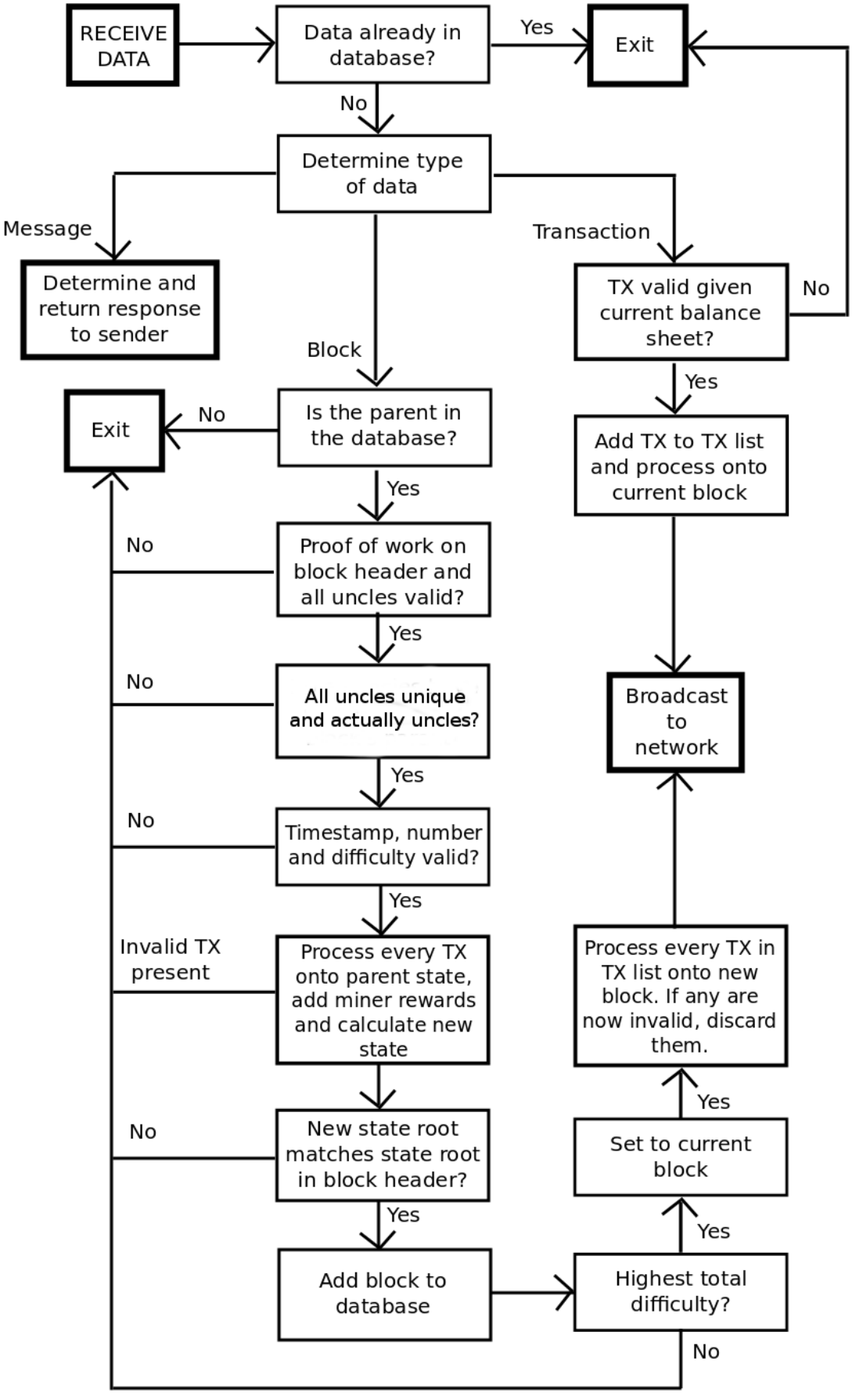
As descrived by Sompolinsky and Zohar, GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale descendants of the block's ancestor (in Ethereum jargon, "uncles") are added to the calculation of which block has the largest total proof of work backing it. To solve the second issue of centralization bias, we go beyond the protocol described by Sompolinsky and Zohar, and also provide block rewards to stales: a stale block receives 87.5% of its base reward, and the nephew that includes the stale block receives the remaining 12.5%. Transaction fees, however, are not awarded to uncles.

Ethereum implements a simplified version of GHOST which only goes down one level. Specifically, a stale block can only be included as an uncle by the direct child of one of its direct siblings, and not any block with a more distant relation. This was done for several reasons. First, unlimited GHOST would include too many

complications into the calculation of which uncles for a given block are valid. Second, unlimited GHOST with compensation as used in Ethereum removes the incentive for a miner to mine on the main chain and not the chain of a public attacker. Finally, calculations show that single-level GHOST has over 80% of the benefit of unlimited GHOST, and provides a stale rate comparable to the 2.5 minute Litecoin even with a 40-second block time. However, we will be conservative and still retain a Primecoin-like 60-second block time because individual blocks may take a longer time to verify.

Ethereum Client P2P Protocol

P2P Protocol



The Ethereum client P2P protocol is a fairly standard cryptocurrency protocol, and can just as easily be used for any other cryptocurrency; the only modification is the introduction of the GHOST protocol described above. The Ethereum client will be mostly reactive; if not provoked, the only thing the client will do by itself is have the networking daemon maintain connections and periodically send a message asking for blocks whose parent is the current block. However, the client will also be more powerful. Unlike bitcoind, which only stores a limited amount of data about the blockchain, the Ethereum client will also act as a fully functional backend for a block explorer.

When the client reads a message, it will perform the following steps:

1. Hash the data, and check if the data with that hash has already been received. If so, exit.
2. Determine the data type. If the data is a transaction, if the transaction is valid add it to the local transaction list, process it onto the current block and publish it to the network. If the data item is a message, respond to it. If the data item is a block, go to step 3.
3. Check if the parent of the block is already stored in the database. If it is not, exit.
4. Check if the proof of work on the block header and all block headers in the "uncle list" is valid. If any are not, exit.
5. Check if every block header in the "uncle list" in the block has the block's parent's parent as its own parent. If any is not, exit. Note that uncle block headers do not need to be in the database; they just need to have the correct parent and a valid proof of work. Also, make sure that uncles are unique and distinct from the parent.
6. Check if the timestamp of the block is at most 15 minutes into the future and ahead of the timestamp of the parent. Check if the difficulty of the block and the block number are correct. If either of these checks fails, exit.
7. Start with the state of the parent of the block, and sequentially apply every transaction in the block to it. At the end, add the miner rewards. If the root hash of the resulting state tree does not match the state root in the block header, exit. If it does, add the block to the database and advance to the next step.
8. Determine `TD(block)` ("total difficulty") for the new block. TD is defined recursively by `TD(genesis_block) = 0` and `TD(B) = TD(B.parent) + sum([u.difficulty for u in B.uncles]) + B.difficulty`. If the new block has higher TD than the current block, set the current block to the new block and continue to the next step. Otherwise, exit.
9. If the new block was changed, apply all transactions in the transaction list to it, discarding from the transaction list any that turn out to be invalid, and rebroadcast the block and those transactions to the network.

The "current block" is a pointer maintained by each node that refers to the block that the node deems as representing the current official state of the network. All messages asking for balances, contract states, etc, have their responses computed by looking at the current block. If a node is mining, the process is only slightly changed: while

doing all of the above, the node also continuously mines on the current block, using its transaction list as the transaction list of the block.

Currency and Issuance

The Ethereum network includes its own built-in currency, ether. The main reason for including a currency in the network is twofold. First, like Bitcoin, ether is rewarded to miners so as to incentivize network security. Second, it serves as a mechanism for paying transaction fees for anti-spam purposes. Of the two main alternatives to fees, per-transaction proof of work similar to Hashcash and zero-fee laissez-faire, the former is wasteful of resources and unfairly punitive against weak computers and smartphones and the latter would lead to the network being almost immediately overwhelmed by an infinitely looping "logic bomb" contract. For convenience and to avoid future argument (see the current mBTC/uBTC/satoshi debate), the denominations will be pre-labelled:

	After 1 year	After 5 years
Currency units	1.9X	3.5X
Fundraiser participants	52.6%	28.6%
Fiduciary members and early contributors	11.8%	6.42%
Additional pre-launch allocations	2.63%	1.42%
Reserve	11.8%	6.42%
Miners	21.1%	57.1%

- 1: wei
- 10^3: (unspecified)
- 10^6: (unspecified)
- 10^9: (unspecified)
- 10^{12}: szabo
- 10^{15}: finney
- 10^{18}: ether

This should be taken as an expanded version of the concept of "dollars" and "cents" or "BTC" and "satoshi" that is intended to be future proof. Szabo, finney and ether will likely be used in the foreseeable future, and the other units will be more . "ether" is intended to be the primary unit in the system, much like the dollar or bitcoin. The right to name the 10^3, 10^6 and 10^9 units will be left as a high-level secondary reward for the fundraiser subject to pre-approval from ourselves.

The issuance model will be as follows:

- Ether will be released in a fundraiser at the price of 1000-2000 ether per BTC, with earlier funders getting a better price to compensate for the increased uncertainty of participating at an earlier stage. The minimum funding amount will be 0.01 BTC. Suppose that X ether gets released in this way

- 0.225X ether will be allocated to the fiduciary members and early contributors who substantially participated in the project before the start of the fundraiser. This share will be stored in a time-lock contract; about 40% of it will be spendable after one year, 70% after two years and 100% after 3 years.
- 0.05X ether will be allocated to a fund to use to pay expenses and rewards in ether between the start of the fundraiser and the launch of the currency
- 0.225X ether will be allocated as a long-term reserve pool to pay expenses, salaries and rewards in ether after the launch of the currency
- 0.4X ether will be mined per year forever after that point

Long-Term Inflation Rate (percent)

Despite the linear currency issuance, just like with Bitcoin over time the inflation *rate* nevertheless tends to zero

For example, after five years and assuming no transactions, 28.6% of the ether will be in the hands of the fundraiser participants, 6.42% in the fiduciary member and early contributor pool, 6.42% paid to the reserve pool, and 57.1% will belong to miners. The permanent linear inflation model reduces the risk of what some see as excessive wealth concentration in Bitcoin, and gives individuals living in present and future eras a fair chance to acquire currency units, while at the same time retaining a strong incentive to obtain and hold ether because the inflation "rate" still tends to zero over time (eg. during year 1000001 the money supply would increase from 500001.5 * X to 500002 * X, an inflation rate of 0.0001%). Furthermore, much of the interest in Ethereum will be medium-term; we predict that if Ethereum succeeds it will see the bulk of its growth on a 1-10 year timescale, and supply during that period will be very much limited.

We also theorize that because coins are always lost over time due to carelessness, death, etc, and coin loss can be modeled as a percentage of the total supply per year, that the total currency supply in circulation will in fact eventually stabilize at a value equal to the annual issuance divided by the loss rate (eg. at a loss rate of 1%, once the supply reaches 40X then 0.4X will be mined and 0.4X lost every year, creating an equilibrium).

Data Format

All data in Ethereum will be stored in recursive length prefix encoding, which serializes arrays of strings of arbitrary length and dimension into strings. For example, `['dog', 'cat']` is serialized (in byte array format) as `[ 130, 67, 100, 111, 103, 67, 99, 97, 116]`; the general idea is to encode the data type and length in a single byte followed by the actual data (eg. converted into a byte array, 'dog' becomes `[ 100, 111, 103 ]`, so its serialization is `[ 67, 100, 111, 103 ]`. Note that RLP encoding is, as suggested by the name, recursive; when RLP encoding an array,

one is really encoding a string which is the concatenation of the RLP encodings of each of the elements. Additionally, note that block number, timestamp, difficulty, memory deposits, account balances and all values in contract storage are integers, and Patricia tree hashes, root hashes, addresses, transaction list hashes and all keys in contract storage are strings. The main difference between the two is that strings are stored as fixed-length data (20 bytes for addresses, 32 bytes for everything else), and integers take up only as much space as they need. Integers are stored in big-endian base 256 format (eg. 32767 in byte array format as `[ 127, 255 ]`).

A full block is stored as:

```
[
    block_header,
    transaction_list,
    uncle_list
]
```

Where:

```
transaction_list = [
    transaction 1,
    transaction 2,
    ...
]

uncle list = [
    uncle_block_header_1,
    uncle_block_header_2,
    ...
]

block_header = [
    parent hash,
    sha3(rlp_encode(uncle_list)),
    coinbase address,
    state_root,
    sha3(rlp_encode(transaction_list)),
    difficulty,
    timestamp,
    extra_data,
    nonce
]
```

Each transaction and uncle block header is itself a list. The data for the proof of work is the RLP encoding of the block WITHOUT the nonce. `uncle_list` and `transaction_list` are the lists of the uncle block headers and transactions in the block, respectively. `nonce` and `extra_data` are both limited to a maximum of 32 bytes, except the genesis block where the `extra_data` parameter will be much larger.

The `state_root` is the root of a Merkle Patricia tree containing (key, value) pairs for all accounts where each address is represented as a 20-byte binary string. At the address of each account, the value stored in the Merkle Patricia tree is a string which is the RLP-serialized form of an object of the form:

```
[ balance, nonce, contract_root, storage_deposit ]
```

The `nonce` is the number of transactions made from the account, and is incremented every time a transaction is made. The purpose of this is to (1) make each transaction valid only once to prevent replay attacks, and (2) to make it impossible (more precisely, cryptographically infeasible) to construct a contract with the same hash as a pre-existing contract. `balance` refers to the account's balance, denominated in wei. `contract_root` is the root of yet another Patricia tree, containing the contract's memory, if that account is controlled by a contract. If an account is not controlled by a contract, the contract root will simply be the empty string. `storage_deposit` is a counter that stores paid storage fees; its function will be discussed in more detail further in this paper.

Mining algorithm

One highly desirable property in mining algorithms is resistance to optimization through specialized hardware. Originally, Bitcoin was conceived as a highly democratic currency, allowing anyone to participate in the mining process with a CPU. In 2010, however, much faster miners exploiting the rapid parallelization offered by graphics processing units (GPUs) rapidly took over, increasing network hashpower by a factor of 100 and leaving CPUs essentially in the dust. In 2013, a further category of specialized hardware, application-specific integrated circuits (ASICs) outcompeted the GPUs in turn, achieving another 100x speedup by using chips fabricated for the sole purpose of computing SHA256 hashes. Today, it is virtually impossible to mine without first purchasing a mining device from one of these companies, and some people are concerned that in 5-10 years' time mining will be entirely dominated by large centralized corporations such as AMD and Intel.

To date, the main way of achieving this goal has been "memory-hardness", constructing proof of work algorithms that require not only a large number of computations, but also a large amount of memory, to validate, thereby making highly parallelized specialized hardware implementations less effective. There have been several implementations of memory-hard proof of work, all of which have their flaws:

- **Scrypt** - Scrypt is a function which is designed to take 128 KB of memory to compute. The algorithm essentially works by filling a memory array with hashes, and then computing intermediate values and finally a result based on the values in the memory array. However, the 128 KB parameter is a very weak threshold, and ASICs for Litecoin are already under development. Furthermore, there is a natural limit to how much memory hardness with Scrypt can be tweaked up to achieve, as the verification process takes just as much memory, and just as much computation, as one round of the mining process.
- **Birthday attacks** - the idea behind birthday-based proofs of work is simple: find values `xn,i,j` such that `i < k, j < k` and `|H(data+xn+i) - H(data+xn+j)| < 2^256 / d^2`. The `d` parameter sets the computational difficulty of finding a block, and the `k` parameter sets the memory hardness. Any birthday algorithm must somehow store all computations of `H(data+xn+i)` in memory so that future

computations can be compared against them. Here, computation is memory-hard, but verification is memory-easy, allowing for extreme memory hardness without compromising the ease of verification. However, the algorithm is problematic for two reasons. First, there is a time-memory tradeoff attack where users 2x less memory can compensate with 2x more computational power, so its memory hardness is not absolute. Second, it may be easy to build specialized hardware devices for the problem, especially once one moves beyond traditional chip and processor architecture and into various classes of hardware-based hash tables or probabilistic analog computing.

- **Dagger** - the idea behind Dagger, an in-house algorithm developed by the Ethereum team, is to have an algorithm that is similar to Scrypt, but which is specially designed so that each individual nonce only depends on a small portion of the data tree that gets built up for each group of ~10 million nonces. Computing nonces with any reasonable level of efficiency requires building up the entire tree, taking up over 100 MB of memory, whereas verifying a nonce only takes about 100 KB. However, Dagger-style algorithms are vulnerable to devices that have multiple computational circuits sharing the same memory, and although this threat can be mitigated it is arguably impossible to fully remove.

As a default, we are currently considering a Dagger-like algorithm with tweaked parameters to minimize specialized hardware attacks, perhaps together with a proof of stake algorithm such as our own Slasher for added security if deemed necessary. However, in order to come up with a proof-of-work algorithm that is better than all existing competitors, our intention is to use some of the funds raised in the fundraiser to host a contest, similar to those used to determine the algorithm for the Advanced Encryption Standard (AES) in 2005 and the SHA3 hash algorithm in 2013, where research groups from around the world compete to develop ASIC-resistant mining algorithms, and have a selection process with multiple rounds of judging determine the winners. The contest will have prizes, and will be open-ended; we encourage research into memory-hard proofs of work, self-modifying proofs of work, proofs of work based on x86 instructions, multiple proofs of work with a human-driven incentive-compatible economic protocol for swapping one out in the future, and any other design that accomplishes the task. There will be opportunities to explore alternatives such as proof of stake, proof of burn and proof of excellence as well.

Transactions

A transaction is stored as:

```
[ nonce, receiving_address, value, [ data item 0, data item 1 ... data item n ], v, r, s ]
```

`nonce` is the number of transactions already sent by that account, encoded in binary form (eg. `0 -> ''`, `7 -> '\x07'`, `1000 -> '\x03\xd8'`). `(v,r,s)` is the raw Electrum-style signature of the transaction without the signature made with the private key corresponding to the sending account, with `0 <= v <= 3`. From an Electrum-style

signature (65 bytes) it is possible to extract the public key, and thereby the address, directly. A valid transaction is one where (i) the signature is well-formed (ie. `0 <= v <= 3, 0 <= r < P, 0 <= s < N, 0 <= r < P - N` if `v >= 2`), and (ii) the sending account has enough funds to pay the fee and the value. A valid block cannot contain an invalid transaction; however, if a contract generates an invalid transaction that transaction will simply have no effect. Transaction fees will be included automatically. If one wishes to voluntarily pay a higher fee, one is always free to do so by constructing a contract which forwards transactions but automatically sends a certain amount or percentage to the miner of the current block.

Transactions sent to the empty string as an address are a special type of transaction, creating a "contract".

Difficulty adjustment

Difficulty is adjusted by the formula:

```
D(genesis_block) = 2^36
D(block) =
    if anc(block,1).timestamp >= anc(block,501).timestamp + 60 * 500:
D(block.parent) - floor(D(block.parent) / 1000)
    else:
D(block.parent) + floor(D(block.parent) / 1000)
```

`anc(block,n)` is the nth generation ancestor of the block; all blocks before the genesis block are assumed to have the same timestamp as the genesis block. This stabilizes around a block time of 60 seconds automatically. The choice of 500 was made in order to balance the concern that for smaller values miners with sufficient hashpower to often produce two blocks in a row would have the incentive to provide an incorrect timestamp to maximize their own reward and the fact that with higher values the difficulty oscillates too much; with the constant of 500, simulations show that a constant hashpower produces a variance of about +/-20%.

Block Rewards

A miner receives three kinds of rewards: a static block reward for producing a block, fees from transactions, and nephew/uncle rewards as described in the GHOST section above. The miner will receive 100% of the block reward for themselves, but transaction fee rewards will be split, so that 50% goes to the miner and the remaining 50% is evenly split among the last 64 miners. The reason for this is to prevent a miner from being able to create an Ethereum block with an unlimited number of operations, paying all transaction fees to themselves, while still maintaining an incentive for miners to include transactions. As described in the GHOST section, uncles only receive 87.5% of their block reward, with the remaining 12.5% going to the including nephew; the transaction fees from the stale block do not go to anyone.

Contracts

In Ethereum, there are two types of entities that can generate and receive transactions: actual people (or bots, as cryptographic protocols cannot distinguish between the two) and contracts. A contract is essentially an automated agent that lives on the Ethereum network, has an Ethereum address and balance, and can send and receive transactions. A contract is "activated" every time someone sends a transaction to it, at which point it runs its code, perhaps modifying its internal state or even sending some transactions, and then shuts down. The "code" for a contract is written in a special-purpose low-level language consisting of a stack, which is not persistent, 2^{256} memory entries, which are also not persistent, and 2^{256} storage entries which constitute the contract's permanent state. Note that Ethereum users will not need to code in this low-level stack language; we will provide a simple C-like language with variables, expressions, conditionals, arrays and while loops, and provide a compiler down to Ethereum script code.

Applications

Here are some examples of what can be done with Ethereum contracts, with all code examples written in our C-like language. The variables `tx.sender`, `tx.value`, `tx.fee`, `tx.data` and `tx.datan` are properties of the incoming transaction, `contract.storage`, and `contract.address` of the contract itself, and `block.contract_storage`, `block.account_balance`, `block.number`, `block.difficulty`, `block.parenthash`, `block.basefee` and `block.timestamp` properties of the block. `block.basefee` is the "base fee" which all transaction fees in Ethereum are calculated as a multiple of; for more info see the "fees" section below. All variables expressed as capital letters (eg. `A`) are constants, to be replaced by actual values by the contract creator when actually releasing the contract.

Sub-currencies

Sub-currencies have many applications ranging from currencies representing assets such as USD or gold to company stocks and even currencies with only one unit issued to represent collectibles or smart property. Advanced special-purpose financial protocols sitting on top of Ethereum may also wish to organize themselves with an internal currency. Sub-currencies are surprisingly easy to implement in Ethereum; this section describes a fairly simple contract for doing so.

The idea is that if someone wants to send `X` currency units to account `A` in currency contract `C`, they will need to make a transaction of the form `(C, 100 * block.basefee, [A, X])`, and the contract parses the transaction and adjusts balances accordingly. For a transaction to be valid, it must send 100 times the base fee worth of ether to the contract in order to "feed" the contract (as each computational step after the first 16 for any contract costs the contract a small fee and the contract will stop working if its balance drains to zero).

```
if tx.value < 100 * block.basefee:
    stop
elif contract.storage[1000]:
    from = tx.sender
```

```
    to = tx.data[0]
    value = tx.data[1]
    if to <= 1000:
        stop
    if contract.storage[from] < value:
        stop
    contract.storage[from] = contract.storage[from] - value
    contract.storage[to] = contract.storage[to] + value
else:
    contract.storage[mycreator] = 10^18
    contract.storage[1000] = 1
```

Ethereum sub-currency developers may also wish to add some other more advanced features:

- Include a mechanism by which people can buy currency units in exchange for ether, perhaps auctioning off a set number of units every day.
- Allow transaction fees to be paid in the internal currency, and then refund the ether transaction fee to the sender. This solves one major problem that all other "sub-currency" protocols have had to date: the fact that sub-currency users need to maintain a balance of sub-currency units to use and units in the main currency to pay transaction fees in. Here, a new account would need to be "activated" once with ether, but from that point on it would not need to be recharged.
- Allow for a trust-free decentralized exchange between the currency and ether. Note that trust-free decentralized exchange between any two contracts is theoretically possible in Ethereum even without special support, but special support will allow the process to be done about ten times more cheaply.

Financial derivatives

The underlying key ingredient of a financial derivative is a data feed to provide the price of a particular asset as expressed in another asset (in Ethereum's case, the second asset will usually be ether). There are many ways to implement a data feed; one method, pioneered by the developers of Mastercoin, is to include the data feed in the blockchain. Here is the code:

```
if tx.sender != FEEDOWNER:
    stop
contract.storage[data[0]] = data[1]
```

Any other contract will then be able to query index `I` of data store `D` by using `block.contract_storage(D)[I]`. A more advanced way to implement a data feed may be to do it off-chain - have the data feed provider sign all values and require anyone attempting to trigger the contract to include the latest signed data, and then use Ethereum's internal scripting functionality to verify the signature. Pretty much any derivative can be made from this, including leveraged trading, options, and even more advanced constructions like collateralized debt obligations (no bailouts here though, so be mindful of black swan risks).

To show an example, let's make a hedging contract. The basic idea is that the contract is created by party A, who puts up 4000 ether as a deposit. The contract then lies open for any party to accept it by putting in 1000 ether. Say that 1000 ether is worth $25 at the time the contract is made, according to index I of data store D. If party B accepts it, then after 30 days anyone can send a transaction to make the contract process, sending the same dollar value worth of ether (in our example, $25) back to B and the rest to A. B gains the benefit of being completely insulated against currency volatility risk without having to rely on any issuers. The only risk to B is if the value of ether falls by over 80% in 30 days - and even then, if B is online B can simply quickly hop onto another hedging contract. The benefit to A is the implicit 0.2% fee in the contract, and A can hedge against losses by separately holding USD in another location (or, alternatively, A can be an individual who is optimistic about the future of Ethereum and wants to hold ether at 1.25x leverage, in which case the fee may even be in B's favor).

```
if tx.value < 200 * block.basefee:
    stop
if contract.storage[1000] == 0:
    if tx.value < 1000 * 10^18:
        stop
    contract.storage[1000] = 0
    contract.storage[1001] = 998 * block.contract_storage(D)[I]
    contract.storage[1002] = block.timestamp + 30 * 86400
    contract.storage[1003] = tx.sender
else:
    ethervalue = contract.storage[1001] / block.contract_storage(D)[I]
    if ethervalue >= 5000 * 10^18:
        mktx(contract.storage[1003],5000 * 10^18,0,0)
    else if block.timestamp > contract.storage[1002]:
        mktx(contract.storage[1003],ethervalue,0,0)
        mktx(A,5000 - ethervalue,0,0)
```

More advanced financial contracts are also possible; complex multi-clause options (eg. "Anyone, hereinafter referred to as X, can claim this contract by putting in 2 USD before Dec 1. X will have a choice on Dec 4 between receiving 1.95 USD on Dec 29 and the right to choose on Dec 11 between 2.20 EUR on Dec 28 and the right to choose on Dec 18 between 1.20 GBP on Dec 30 and paying 1 EUR and getting 3.20 EUR on Dec 29") can be defined simply by storing a state variable just like the contract above but having more clauses in the code, one clause for each possible state. Note that financial contracts of any form do need to be fully collateralized; the Ethereum network controls no enforcement agency and cannot collect debt.

Identity and Reputation Systems

The earliest alternative cryptocurrency of all, Namecoin, attempted to use a Bitcoin-like blockchain to provide a name registration system, where users can register their names in a public database alongside other data. The major cited use case is for a DNS system, mapping domain names like "bitcoin.org" (or, in Namecoin's case, "bitcoin.bit") to an IP address. Other use cases include email authentication and potentially more advanced reputation systems. Here is a simple contract to provide a Namecoin-like name registration system on Ethereum:

```
if tx.value < block.basefee * 200:
    stop
if contract.storage[tx.data[0]] or tx.data[0] < 100:
    stop
contract.storage[tx.data[0]] = tx.data[1]
```

One can easily add more complexity to allow users to change mappings, automatically send transactions to the contract and have them forwarded, and even add reputation and web-of-trust mechanics.

Decentralized Autonomous Organizations

The general concept of a "decentralized autonomous organization" is that of a virtual entity that has a certain set of members or shareholders which, perhaps with a 67% majority, have the right to spend the entity's funds and modify its code. The members would collectively decide on how the organization should allocate its funds. Methods for allocating a DAO's funds could range from bounties, salaries to even more exotic mechanisms such as an internal currency to reward work. This essentially replicates the legal trappings of a traditional company or nonprofit but using only cryptographic blockchain technology for enforcement. So far much of the talk around DAOs has been around the "capitalist" model of a "decentralized autonomous corporation" (DAC) with dividend-receiving shareholders and tradable shared; an alternative, perhaps described as a "decentralized autonomous community", would have all members have an equal share in the decision making and require 67% of existing members to agree to add or remove a member. The requirement that one person can only have one membership would then need to be enforced collectively by the group.

Some "skeleton code" for a DAO might look as follows.

There are three transaction types:

- `[0,k]` to register a vote in favor of a code change
- `[1,k,L,v0,v1...vn]` to register a code change at code `k` in favor of setting memory starting from location `L` to `v0, v1 ... vn`
- `[2,k]` to finalize a given code change

Note that the design relies on the randomness of addresses and hashes for data integrity; the contract will likely get corrupted in some fashion after about `2^128` uses, but that is acceptable since nothing close to that volume of usage will exist in the foreseeable future. `2^255` is used as a magic number to store the total number of members, and a membership is stored with a `1` at the member's address. The last three lines of the contract are there to add `C` as the first member; from there, it will be `C`'s responsibility to use the democratic code change protocol to add a few other members and code to bootstrap the organization.

```
if tx.value < tx.basefee * 200:
    stop
if contract.storage[tx.sender] == 0:
    stop
k = sha3(32,tx.data[1])
```

```
if tx.data[0] == 0:
    if contract.storage[k + tx.sender] == 0:
        contract.storage[k + tx.sender] = 1
        contract.storage[k] += 1
else if tx.data[0] == 1:
    if tx.value <= tx.datan * block.basefee * 200 or contract.storage[k]:
        stop
    i = 2
    while i < tx.datan:
        contract.storage[k + i] = tx.data[i]
        i = i + 1
    contract.storage[k] = 1
    contract.storage[k+1] = tx.datan
else if tx.data[0] == 2:
    if contract.storage[k] >= contract.storage[2 ^ 255] * 2 / 3:
        if tx.value <= tx.datan * block.basefee * 200:
            stop
        i = 3
        L = contract.storage[k+1]
        loc = contract.storage[k+2]
        while i < L:
            contract.storage[loc+i-3] = tx.data[i]
            i = i + 1
if contract.storage[2 ^ 255 + 1] == 0:
    contract.storage[2 ^ 255 + 1] = 1
    contract.storage[C] = 1
```

This implements the "egalitarian" DAO model where members have equal shares. One can easily extend it to a shareholder model by also storing how many shares each owner holds and providing a simple way to transfer shares.

DAOs and DACs have already been the topic of a large amount of interest among cryptocurrency users as a future form of economic organization, and we are very excited about the potential that DAOs can offer. In the long term, the Ethereum fund itself intends to transition into being a fully self-sustaining DAO.

Further Applications

1) **Savings wallets**. Suppose that Alice wants to keep her funds safe, but is worried that she will lose or someone will hack her private key. She puts ether into a contract with Bob, a bank, as follows: Alice alone can withdraw a maximum of 1% of the funds per day, Alice and Bob together can withdraw everything, and Bob alone can withdraw a maximum of 0.05% of the funds. Normally, 1% per day is enough for Alice, and if Alice wants to withdraw more she can contact Bob for help. If Alice's key gets hacked, she runs to Bob to move the funds to a new contract. If she loses her key, Bob will get the funds out eventually. If Bob turns out to be malicious, she can still withdraw 20 times faster than he can.

2) **Crop insurance**. One can easily make a financial derivatives contract but using a data feed of the weather instead of any price index. If a farmer in Iowa purchases a derivative that pays out inversely based on the precipitation in Iowa, then if there is a drought, the farmer will automatically receive money and if there is enough rain the farmer will be happy because their crops would do well.

3) A **decentrally managed data feed**, using proof-of-stake voting to give an average (or more likely, median) of everyone's opinion on the price of a commodity, the weather or any other relevant data.

4) **Smart multisignature escrow**. Bitcoin allows multisignature transaction contracts where, for example, three out of a given five keys can spend the funds. Ethereum allows for more granularity; for example, four out of five can spend everything, three out of five can spend up to 10% per day, and two out of five can spend up to 0.5% per day. Additionally, Ethereum multisig is asynchronous - two parties can register their signatures on the blockchain at different times and the last signature will automatically send the transaction.

5) **Peer-to-peer gambling**. Any number of peer-to-peer gambling protocols, such as Frank Stajano and Richard Clayton's Cyberdice, can be implemented on the Ethereum blockchain. The simplest gambling protocol is actually simply a contract for difference on the next block hash. From there, entire gambling services such as SatoshiDice can be replicated on the blockchain either by creating a unique contract per bet or by using a quasi-centralized contract.

6) A full-scale **on-chain stock market**. Prediction markets are also easy to implement as a trivial consequence.

7) An **on-chain decentralized marketplace**, using the identity and reputation system as a base.

8) **Decentralized Dropbox**. One setup is to encrypt a file, build a Merkle tree out of it, put the Merkle root into a contract alongside a certain quantity of ether, and distribute the file across some secondary network. Every day, the contract would randomly select a branch of the Merkle tree depending on the block hash, and give X ether to the first node to provide that branch to the contract, thereby encouraging nodes to store the data for the long term in an attempt to earn the prize. If one wants to download any portion of the file, one can use a micropayment channel-style contract to download the file from a few nodes a block at a time.

How do contracts work?

A contract making transaction is encoded as follows:

```
[
    nonce,
    '',
    value,
    [
        data item 0,
        data item 1,
        ...
    ],
    v,
    r,
    s
```

```
]
```

The data items will, in most cases, be script codes (more on this below). Contract creation transaction validation happens as follows:

1. Deserialize the transaction, and extract its sending address from its signature.
2. Calculate the transaction's fee as `NEWCONTRACTFEE` plus storage fees for the code. Check that the balance of the creator is at least the transaction value plus the fee. If not, exit.
3. Take the last 20 bytes of the sha3 hash of the RLP encoding of the transaction making the contract. If an account with that address already exists, exit. Otherwise, create the contract at that address
4. Copy data item `i` to storage slot `i` in the contract for all `i` in `[0 ... n-1]` where `n` is the number of data items in the transaction, and initialize the contract with the transaction's value as its value. Subtract the value and fee from the creator's balance.

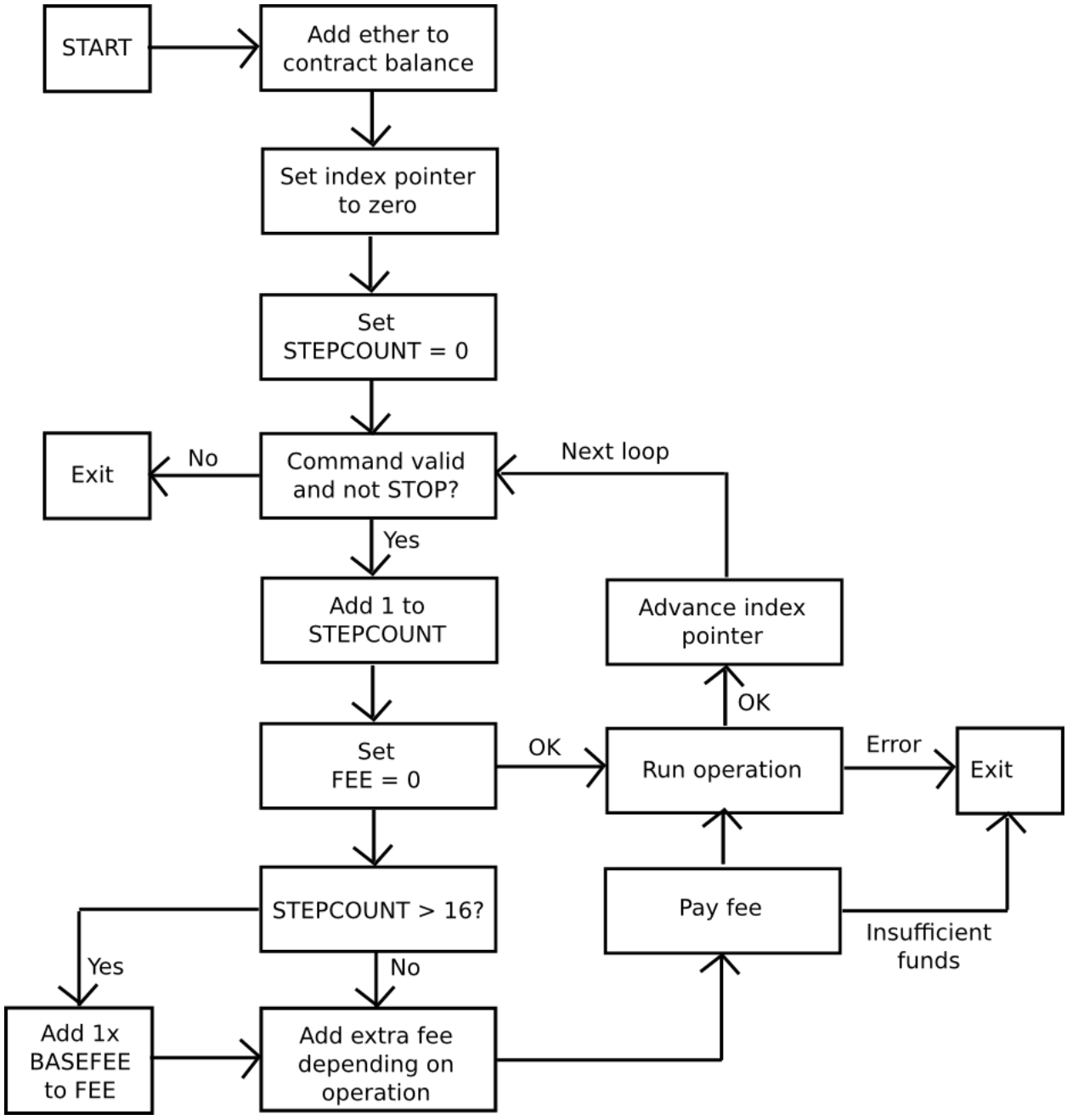
Language Specification

The contract scripting language is a hybrid of assembly language and Bitcoin's stack-based language, maintaining an index pointer that usually increments by one after every operation and continuously processing the operation found at the current index pointer. All opcodes are numbers in the range `[0 ... 63]`; labels further in this description such as `STOP`, `EXTRO` and `BALANCE` refer to specific values are defined further below. The scripting language has access to three kinds of memory:

- **Stack** - a form of temporary storage that is reset to an empty list every time a contract is executed. Operations typically add and remove values to and from the top of the stack, so the total length of the stack will shrink and grow over the course of the program's execution.
- **Memory** - a temporary key/value store that is reset to containing all zeroes every time a contract is executed. Keys and values in memory are integers in the range `[0 ... 2^256-1]`
- **Storage** - a persistent key/value store that is initially set to contain all zeroes, except for some script code inserted at the beginning when the contract is created as described above. Keys and values in storage are integers in the range `[0 ... 2^256-1]`

Whenever a transaction is sent to a contract, the contract executes its scripting code. The precise steps that happen when a contract receives a transaction are as follows:

Contract Script Interpretation



1. The contract's ether balance increases by the amount sent
2. The index pointer is set to zero, and `STEPCOUNT = 0`
3. Repeat forever:
    - if the command at the index pointer is `STOP`, invalid or greater than 63, exit from the loop
    - set `MINERFEE = 0`, `VOIDFEE = 0`
    - set `STEPCOUNT <- STEPCOUNT + 1`
    - if `STEPCOUNT > 16`, set `MINERFEE <- MINERFEE + STEPFEE`
    - see if the command is `LOAD` or `STORE`. If so, set `MINERFEE <- MINERFEE + DATAFEE`
    - see if the command will modify a storage field, say modifying `KEY` from `OLDVALUE` to `NEWVALUE`. Let `F(K,V)` be `0` if `V ==`

`0` else `(len(K) + len(V)) * STORAGEFEE` in bytes. Set `VOIDFEE <- VOIDFEE - F(KEY,OLDVALUE) + F(KEY,NEWVALUE)`. Computing `len(K)` ignores leading zero bytes.
    - see if the command is `EXTRO` or `BALANCE`. If so, set `MINERFEE <- MINERFEE + EXTROFEE`
    - see if the command is a crypto operation. If so, set `MINERFEE <- MINERFEE + CRYPTOFEE`
    - if `MINERFEE + VOIDFEE > CONTRACT.BALANCE`, HALT and exit from the loop
    - subtract `MINERFEE` from the contract's balance and add `MINERFEE` to a running counter that will be added to the miner's balance once all transactions are parsed.
    - set `DELTA = max(-CONTRACT.STORAGE_DEPOSIT,VOIDFEE)` and `CONTRACT.BALANCE <- CONTRACT.BALANCE - DELTA` and `CONTRACT.STORAGE_DEPOSIT <- CONTRACT.STORAGE_DEPOSIT + DELTA`. Note that `DELTA` can be positive or negative; the only restriction is that the contract's deposit cannot go below zero.
    - run the command
    - if the command did not exit with an error, update the index pointer and return to the start of the loop. If the contract did exit with an error, break out of the loop. Note that a contract exiting with an error does not make the transaction or the block invalid; it simply means that the contract execution halts midway through.

In the following descriptions, `S[-1]`, `S[-2]`, etc represent the topmost, second topmost, etc items on the stack. The individual opcodes are defined as follows:

- `(0) STOP` - halts execution
- `(1) ADD` - pops two items and pushes `S[-2] + S[-1] mod 2^256`
- `(2) MUL` - pops two items and pushes `S[-2] * S[-1] mod 2^256`
- `(3) SUB` - pops two items and pushes `S[-2] - S[-1] mod 2^256`
- `(4) DIV` - pops two items and pushes `floor(S[-2] / S[-1])`. If `S[-1] = 0`, halts execution.
- `(5) SDIV` - pops two items and pushes `floor(S[-2] / S[-1])`, but treating values above `2^255 - 1` as negative (ie. `x -> 2^256 - x`). If `S[-1] = 0`, halts execution.
- `(6) MOD` - pops two items and pushes `S[-2] mod S[-1]`. If `S[-1] = 0`, halts execution.
- `(7) SMOD` - pops two items and pushes `S[-2] mod S[-1]`, but treating values above `2^255 - 1` as negative (ie. `x -> 2^256 - x`). If `S[-1] = 0`, halts execution.
- `(8) EXP` - pops two items and pushes `S[-2] ^ S[-1] mod 2^256`
- `(9) NEG` - pops one item and pushes `2^256 - S[-1]`
- `(10) LT` - pops two items and pushes 1 if `S[-2] < S[-1]` else 0
- `(11) LE` - pops two items and pushes 1 if `S[-2] <= S[-1]` else 0
- `(12) GT` - pops two items and pushes 1 if `S[-2] > S[-1]` else 0
- `(13) GE` - pops two items and pushes 1 if `S[-2] >= S[-1]` else 0

- (14) EQ - pops two items and pushes 1 if S[-2] == S[-1] else 0
- (15) NOT - pops one item and pushes 1 if S[-1] == 0 else 0
- (16) MYADDRESS - pushes the contract's address as a number
- (17) TXSENDER - pushes the transaction sender's address as a number
- (18) TXVALUE - pushes the transaction value
- (19) TXDATAN - pushes the number of data items
- (20) TXDATA - pops one item and pushes data item S[-1], or zero if index out of range
- (21) BLK_PREVHASH - pushes the hash of the previous block (NOT the current one since that's impossible!)
- (22) BLK_COINBASE - pushes the coinbase of the current block
- (23) BLK_TIMESTAMP - pushes the timestamp of the current block
- (24) BLK_NUMBER - pushes the current block number
- (25) BLK_DIFFICULTY - pushes the difficulty of the current block
- (26) BLK_NONCE - pushes the nonce of the current block
- (27) BASEFEE - pushes the base fee (x as defined in the fee section below)
- (32) SHA256 - pops two items, and then constructs a string by taking the ceil(S[-1] / 32) items in memory from index S[-2] to (S[-2] + ceil(S[-1] / 32) - 1) mod 2^256, prepending zero bytes to each one if necessary to get them to 32 bytes, and takes the last S[-1] bytes. Pushes the SHA256 hash of the string
- (33) RIPEMD160 - works just like SHA256 but with the RIPEMD-160 hash
- (34) ECMUL - pops three items. If (S[-2],S[-1]) are a valid point in secp256k1, including both coordinates being less than P, pushes (S[-2],S[-1]) * S[-3], using (0,0) as the point at infinity. Otherwise, pushes (2^256 - 1, 2^256 - 1). Note that there are no restrictions on S[-3]
- (35) ECADD - pops four items and pushes (S[-4],S[-3]) + (S[-2],S[-1]) if both points are valid, otherwise (2^256 - 1,2^256 - 1)
- (36) ECSIGN - pops two items and pushes (v,r,s) as the Electrum-style RFC6979 deterministic signature of message hash S[-1] with private key S[-2] mod N with 0 <= v <= 3
- (37) ECRECOVER - pops four items and pushes (x,y) as the public key from the signature (S[-3],S[-2],S[-1]) of message hash S[-4]. If the signature has invalid v,r,s values (ie. v not in [27,28], r not in [0,P], s not in [0,N]), return (2^256 - 1,2^256 - 1)
- (38) ECVALID - pops two items and pushes 1 if (S[-2],S[-1]) is a valid secp256k1 point (including (0,0)) else 0
- (39) SHA3 - works just like SHA256 but with the SHA3 hash, 256 bit version
- (48) PUSH - pushes the item in memory at the index pointer + 1, and advances the index pointer by 2.
- (49) POP - pops one item.
- (50) DUP - pushes S[-1] to the stack.
- (51) SWAP - pops two items and pushes S[-1] then S[-2]
- (52) MLOAD - pops two items and sets the item in memory at index S[-1] to S[-2]
- (53) MSTORE - pops two items and sets the item in memory at index S[-1] to S[-2]

- `(54) SLOAD` - pops two items and sets the item in storage at index `S[-1]` to `S[-2]`
- `(55) SSTORE` - pops two items and sets the item in storage at index `S[-1]` to `S[-2]`
- `(56) JMP` - pops one item and sets the index pointer to `S[-1]`
- `(57) JMPI` - pops two items and sets the index pointer to `S[-2]` only if `S[-1]` is nonzero
- `(58) IND` - pushes the index pointer
- `(59) EXTRO` - pops two items and pushes memory index `S[-2]` of contract `S[-1]`
- `(60) BALANCE` - pops one item and pushes balance of the account with that address, or zero if the address is invalid
- `(61) MKTX` - pops four items and initializes a transaction to send `S[-2]` ether to `S[-1]` with `S[-3]` data items. Takes items in memory from index `S[-4]` to index `(S[-4] + S[-3] - 1) mod 2^256` as the transaction's data items.
- `(63) SUICIDE` - pops one item, destroys the contract and clears all storage, sending the entire balance plus the contract deposit to the account at `S[-1]`

As mentioned above, the intent is not for people to write scripts directly in Ethereum script code; rather, we will release compilers to generate ES from higher-level languages. The first supported language will likely be the simple C-like language used in the descriptions above, and the second will be a more complete first-class-function language with support for arrays and arbitrary-length strings. Compiling the C-like language is fairly simple as far as compilers go: variables can be assigned a memory index, and compiling an arithmetic expression essentially involves converting it to reverse Polish notation (eg. `(3 + 5) * (x + y) -> PUSH 3 PUSH 5 ADD PUSH 0 MLOAD PUSH 1 MLOAD ADD MUL`). First-class function languages are more involved due to variable scoping, but the problem is nevertheless tractable. The likely solution will be to maintain a linked list of stack frames in memory, giving each stack frame N memory slots where N is the total number of distinct variable names in the program. Variable access will consist of searching down the stack frame list until one frame contains a pointer to the variable, copying the pointer to the top stack frame for memoization purposes, and returning the value at the pointer. However, these are longer term concerns; compilation is separate from the actual protocol, and so it will be possible to continue to research compilation strategies long after the network is set running.

Fees

In Bitcoin, there are no mandatory transaction fees. Transactions can optionally include fees which are paid to miners, and it is up to the miners to decide what fees they are willing to accept. In Bitcoin, such a mechanism is already imperfect; the need for a 1 MB block size limit alongside the fee mechanism shows this all too well. In Ethereum, because of its Turing-completeness, a purely voluntary fee system would be catastrophic. Instead, Ethereum will have a system of mandatory fees, including a transaction fee and six fees for contract computations. The fees are currently set to:

- `TXFEE` (100x) - fee for sending a transaction

- `NEWCONTRACTFEE` (100x) - fee for creating a new contract, not including the storage fee for each item in script code
- `STEPFEE` (1x) - fee for every computational step after than first sixteen in contract execution
- `STORAGEFEE` (5x) - per-byte fee for adding to contract storage. The storage fee is the only fee that is not paid to a miner, and is refunded when storage used by a contract is reduced or removed.
- `DATAFEE` (20x) - fee for accessing or setting a contract's memory from inside that contract
- `EXTROFEE` (40x) - fee for accessing memory from another contract inside a contract
- `CRYPTOFEE` (20x) - fee for using any of the cryptographic operations

The coefficients will be revised as more hard data on the relative computational cost of each operation becomes available. The hardest part will be setting the value of `x`. There are currently two main solutions that we are considering:

- Make `x` inversely proportional to the square root of the difficulty, so `x = floor(10^21 / floor(difficulty ^ 0.5))`. This automatically adjusts fees down as the value of ether goes up, and adjusts fees down as computers get more powerful due to Moore's Law.
- Use proof of stake voting to determine the fees. In theory, stakeholders do not benefit directly from fees going up or down, so their incentives would be to make the decision that would maximize the value of the network.

A hybrid solution is also possible, using proof of stake voting, but with the inverse square root mechanism as an initial policy.

Conclusion

The Ethereum protocol's design philosophy is in many ways the opposite from that taken by many other cryptocurrencies today. Other cryptocurrencies aim to add complexity and increase the number of "features"; Ethereum, on the other hand, takes features away. The protocol does not "support" multisignature transactions, multiple inputs and outputs, hash codes, lock times or many other features that even Bitcoin provides. Instead, all complexity comes from a universal, Turing-complete scripting language, which can be used to build up literally any feature that is mathematically describable through the contract mechanism. As a result, we have a protocol with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, Ethereum is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come.

References and Further Reading

1. Colored coins whitepaper: https://docs.google.com/a/buterin.com/document/d/1AnkP_cVZTCMLIzw4DvsW6M8Q2JC0lIzrTLuoWu2z1BE/edit
2. Mastercoin whitepaper: https://github.com/mastercoin-MSC/spec
3. Decentralized autonomous corporations, Bitcoin Magazine: http://bitcoinmagazine.com/7050/bootstrapping-a-decentralized-autonomous-corporation-part-i/
4. Smart property: https://en.bitcoin.it/wiki/Smart_Property
5. Smart contracts: https://en.bitcoin.it/wiki/Contracts
6. Simplified payment verification: https://en.bitcoin.it/wiki/Scalability#Simplified*payment*verification
7. Merkle trees: http://en.wikipedia.org/wiki/Merkle_tree
8. Patricia trees: http://en.wikipedia.org/wiki/Patricia_tree
9. Bitcoin whitepaper: http://bitcoin.org/bitcoin.pdf
10. GHOST: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf
11. StorJ and Autonomous Agents, Jeff Garzik: http://garzikrants.blogspot.ca/2013/01/storj-and-bitcoin-autonomous-agents.html
12. Mike Hearn on Smart Property at Turing Festival: http://www.youtube.com/watch?v=Pu4PAMFPo5Y
13. Ethereum RLP: http://wiki.ethereum.org/index.php/RLP
14. Ethereum Merkle Patricia trees: http://wiki.ethereum.org/index.php/Patricia_Tree
15. Ethereum Dagger: http://wiki.ethereum.org/index.php/Dagger
16. Ethereum C-like language: http://wiki.ethereum.org/index.php/CLL
17. Ethereum Slasher: http://blog.ethereum.org/?p=39/slasher-a-punitive-proof-of-stake-algorithm
18. Scrypt parameters: https://litecoin.info/User:Iddo/Comparison*between*Litecoin*and*Bitcoin#SHA256*mining*vs*scrypt*mining
19. Litecoin ASICs: https://axablends.com/merchants-accepting-bitcoin/litecoin-discussion/litecoin-scrypt-asic-miners/